10.07.15

RSA Access Manager Agent 5.0 SP3 Web Agent Extension API Guide

The RSA Access Manager Agent 5.0 SP3 (Access Manager Agent) Web Agent Extension (WAX) Application Programming Interface (API) allows developers to extend and customize the functionality of any Access Manager Agent.

This document summarizes the features of the WAX API and details how to create and use a WAX Program.

Contents:

Overview

The Access Manager Agent WAX API allows developers to extend and customize the functionality of any Access Manager Agent. For certain operations such as custom authentication, WAX API functionality can be used instead of Runtime API functionality. For details of the WAX API components, see "WAX API Components" on page 10.

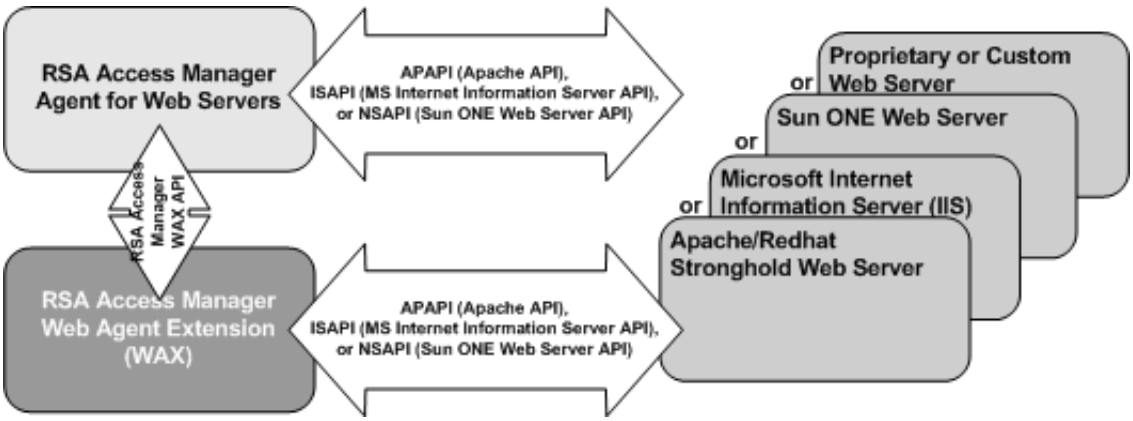
The WAX API is used to create customized phase handlers, which can then be called during URL request processing. A custom phase handler is called a WAX program. For details about how to build a WAX program, see "Build a WAX Program" on page 10.

WAX programs modify the behavior of the Access Manager Agent authentication and authorization processing to perform the following operations:

- Provide custom authentication of users.
- Direct the web server to display a specific HTML file based on the Authorization Server return code. Usually, the return code is a denial of access for a specific reason and the HTML page explains why access was denied.
- Provide custom logging.
- Integrate the Access Manager Agent with proprietary web server Access Manager Agents or third-party Agents.

WAX programs are implemented using a call-back scheme. A WAX program must register itself to the Access Manager Agent and define the various routines to call when processing a URL request. A call made using the WAX API is available to the Access Manager Agent to which it is registered, and to the web server.

This figure illustrates the relationships between a WAX program, an Access Manager Agent, and a web server, through their respective APIs.



The Access Manager Agent is implemented using the APIs of the respective web server vendors. WAX programs also use the APIs of these servers.

URL Processing

During a URL request, the web server invokes the Access Manager Agent to perform authentication and authorization. The Access Manager Agent processes the request by executing a sequence of phases. During each phase, the Access Manager Agent first invokes a phase handler to perform an associated action then invokes a status handler to handle the status from the phase handler. The majority of WAX programs are phase handlers.

Phase Handlers

As the Access Manager Agent processes a phase, it first invokes any custom phase handler that is registered. The custom phase handler performs its action and returns a boolean value indicating whether or not it handled the phase.

- If it returns `TRUE`, the Access Manager Agent goes to the next phase and no additional handlers are invoked for this phase.
- If it returns `FALSE`, the Access Manager Agent either:
  - Invokes the next registered handler for this phase
  - Invokes the default handler for this phase if no other handlers are registered.

For detailed descriptions of the phase handlers and their behavior, see "Phase Handlers" on page 5.

Status Handler

The status handler is invoked after each phase. The status handler manages the processing flow based on the status code returned from any phase handler. It determines the next phase to execute or stops the execution, and returns the status to the web server. Information for each request is passed between the phase handlers and status handler using a hash table. There is a single status handler in the loop, but each phase has its own distinct phase handler.

Status Codes

The status code determines the action and the next phase of execution. For convenience, two macros, `SET_STATUS` and `GET_STATUS`, are supplied to set and get the status respectively.

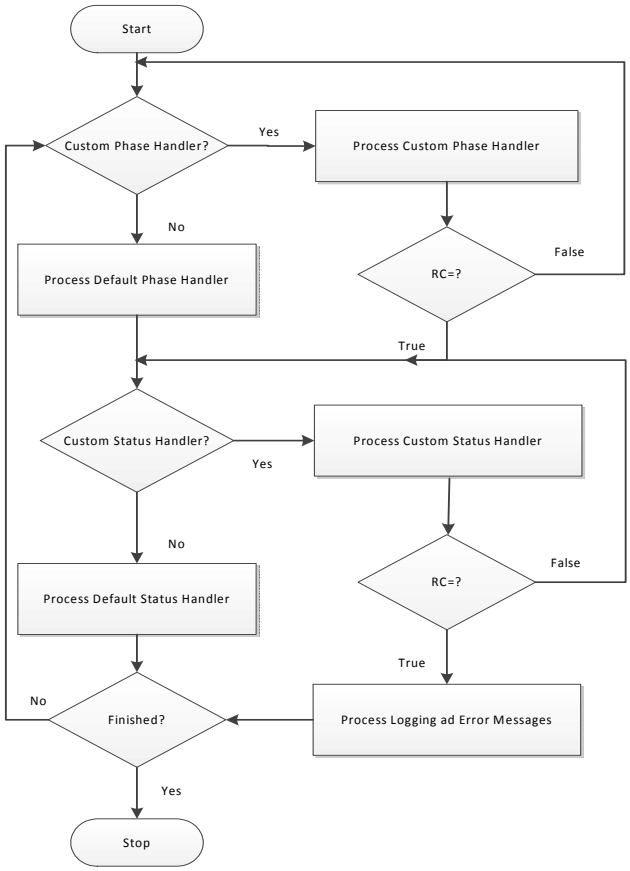
For detailed descriptions of the status handler and the supported status codes, see "Status Handler" on page 9.

WAX Chaining

The use of multiple phase handlers is called WAX chaining. As many phase handlers as required may be chained. In a given WAX chain, only the last phase handler returns `TRUE`; all others return `FALSE`, regardless of the success or failure of their internal actions. This instructs the Access Manager Agent to invoke the next handler in the current phase. After a phase handler has completed its actions, the status handler is invoked to handle the result from the phase handler.

- The Access Manager Agent first calls any custom status handler that may exist. Like the phase handler, the custom status handler returns a boolean value indicating whether or not it handled the status.
- If the custom status handler returns `TRUE`, the default status handlers are invoked.
  - Only logging and messaging actions are performed.
  - Processing for the URL request stops.
- If the custom status handler returns `FALSE`, the next registered handler or default handler is invoked.

The following flowchart illustrates WAX chaining.



Phase Handlers

During the processing of a URL request, the Access Manager Agent executes a sequence of phases to perform authentication, authorization, and single sign-on, ultimately determining the accessibility of the URL.

The phase handlers are listed in the order in which they run:

• Path Check Handler	`CT_PATH_CHECK_HANDLER`
• Session Handler	`CT_SESSION_HANDLER`
• Pre-authentication Handler	`CT_PREAUTHENTICATION_HANDLER`
• Authentication Handler	`CT_AUTHENTICATION_HANDLER`
• Authorization Handler	`CT_ACCESS_HANDLER`
• Cookie Handler	`CT_COOKIE_HANDLER`

In addition, there is a handler called the Status Handler, `CT_STATUS_HANDLER`, that is invoked after each phase. For the recognized status codes and their resulting actions, refer to "Status Handler" on page 9.

Path Check Handler

The path check handler determines whether the requested URL is protected. The handler invokes the RSA Access Manager (Access Manager) Authorization Server to perform the path check.

- If the URL is not protected:
  - The system returns the `CT_AUTH_URL_UNPROTECTED` status code
  - The default status handler instructs the web server to serve the requested URL.
- If the URL is protected:
  - The system returns the `CT_AUTH_URL_PROTECTED` status code
  - The default status handler:
    - Retrieves the list of required and/or allowed authentication types from the Authorization Server
    - Inserts the list of authentication types into the request table under the key `CT_ALLOWABLE_AUTH_MODES`
    - Calls the next handler.

For each protected resource, an administrator sets the allowed and required authentication types in the *webagent.conf* file.

Session Handler

The session handler determines whether or not the cookie, used for single-sign-on support. has expired.

- If the cookie has expired, the phase returns the status code `CT_SESSION_EXPIRED`.
  - With HTTP basic authentication, the default status handler sends a `WWW-Authenticate` response (HTTP 401) to the browser for re-authentication.
  - With form-based log on, the default status handler redirects the request to the log on page determined by `CT_ALLOWABLE_AUTH_MODES` key in the request table.
- If the cookie has not expired, the system returns the status code `CT_SESSION_ACTIVE`, and the next handler is called.

Pre-authentication Handler

The pre-authentication handler gets the allowed and/or required authentication types needed by the URL from the request table (`CT_ALLOWABLE_AUTH_MODES`), and checks that the user has already authenticated with all or some of the authentication types.

- If the user has already authenticated with the required authentication type or types, the pre-authentication handler sets the status code `CT_CHECK_ACCESS_REQUIRED`, causing the authorization handler to be invoked next.
- If there are required authentication types for which the user has not authenticated, the pre-authentication handler asks the appropriate authentication handlers to authenticate the user.

Authentication Handler

The authentication handler may be a standard Access Manager authentication handler or a custom built handler. The following steps describe how the authentication handler performs Access Manager Basic authentication. This behavior provides a good model for building custom authentication handlers.

1. The authentication handler starts with the first, not-yet-satisfied authentication type on the `CT_ALLOWABLE_AUTH_MODES` list.
2. To authenticate the user through form-based authentication, the authentication handler must get the user credential, `CT_USER`/`CT_PASSWORD` or `CT_DN`, from the request table.
   a. If no user ID or DN is present in the request table, the authentication handler sets a status code of `CT_AUTH_BAD_USERNAME`.
   b. If no password is present in the request table, the authentication handler sets a status code of `CT_AUTH_BAD_PASSWORD`.

3. If the user credential exists in the request table, the authentication handler attempts to authenticate the user with the appropriate authentication type.
4. Based on what is returned from the authentication attempt, the authentication handler sets the appropriate status code.

Successful Authentication

If an authentication is successful, the authentication handler sets the authenticated bit, `CT_AUTHENTICATED`, to signify that the user has authenticated successfully with the current authentication type, and therefore does not need to authenticate against this mode in the future. A custom authentication handler must set the `CT_AUTH_CUSTOM` bit of the `CT_AUTHENTICATED` bit-field to indicate that the custom authentication was successful.

Note: The authentication handler **must not** set `CT_AUTH_MODE`; this indicates what authorization type was requested and is set by the pre-authentication handler.

Failed Authentication

When using non-forms-based authentication, the web server is instructed to return a `WWW-Authenticate` response (HTTP 401).

With forms-based authentication, the browser is redirected to the configured error message form. The following return codes can be used to choose an appropriate error page to display to the user.

- `CT_AUTH_BAD_USERNAME`: The user is not defined in the Access Manager database.
- `CT_AUTH_BAD_PASSWORD`: The password specified does not match the user password.

  Note: The system is more secure if only vague error messages display. For example, if a person is trying to break in and receives a “bad password” message, they will know that they have used a valid User ID.

The web server is instructed to return a `FORBIDDEN` response (HTTP 403) or is redirected to a custom error page for the following return codes:

- `CT_AUTH_EXPIRED_ACCOUNT`: The account has expired.
- `CT_AUTH_INACTIVE_ACCOUNT`: The account has not started yet.
- `CT_AUTH_PASSWORD_EXPIRED`: The user password has expired. The password must be reset.
- `CT_AUTH_PASSWORD_EXPIRED_FORCED`: An administrator has expired the user password. The password must be reset.
- `CT_AUTH_PASSWORD_EXPIRED_NEW_USER`: The user is logging on for the first time. The password must be reset.

- `CT_AUTH_USER_LOCKED_OUT`: An administrator has explicitly locked out the user.

Authorization Handler

The authorization handler determines whether or not a user has access to the requested URL. The authorization handler invokes the Authorization Server to perform the authorization checking and sets the status code with the returned value.

- If the Authorization Server returns a `CT_AUTH_URL_ACCESS_DENIED`, the web server is instructed to return a `FORBIDDEN` response (HTTP 403) or is redirected to a custom error page.
- If the Authorization Server returns a `CT_AUTH_URL_ACCESS_ALLOWED`, the authorization handler sets the status code to `CT_CREATE_COOKIE` to instruct the status handler to invoke the cookie handler.

Cookie Handler

The cookie handler creates a cookie to send back to the user with a successful request for a protected URL, and adds URL `REQUEST` related information that is stored in `ct_request_table`.

- If the cookie handler successfully creates a cookie, it sets the status to `CT_AUTH_URL_ACCESS_ALLOWED` and the default status handler directs the web server to provide the requested URL.
- If there was an error creating the cookie, the cookie handler sets a status of `CT_COOKIE_ERROR` and instructs the web server to return a `SERVER ERROR` (HTTP 500) to the browser.

The Access Manager Agent provides users with a 2 KB data buffer within the cookie that can be used for personalization or custom development. This data buffer can be used to provide additional functionality to an Access Manager cookie. For example, to create a WAX program to add an e-mail address or other user attributes to a cookie. Another option is to utilize the cookie for user management functions like encryption. For more information, see “Request Data” on page 19.

Status Handler

The Status Handler, `CT_STATUS_HANDLER`, is invoked after each phase. The status handler manages the processing flow based on the status code returned from any phase handler. The status code is set in `ct_request_data`. The status handler does not know the identity of the phase handler that invokes it. The status handler uses the status code to determine the next action to take or the next phase handler to execute.

For convenience, two macros, `SET_STATUS` and `GET_STATUS`, are supplied to set and get the status respectively.

The status code values are found in the *ct_function_table.h* header file. The following table shows the recognized values and meanings of the different status codes.

Status Code	Resulting Actions
`CT_SESSION_ACTIVE`	Execute authentication phase with active authentication.
`CT_AUTH_URL_PROTECTED`	Execute session phase.
`CT_CHECK_ACCESS_REQUIRED`	Execute Authorization phase.
`CT_CREATE_COOKIE`	Execute Cookie phase.
`CT_AUTH_URL_ACCESS_ALLOWED` `CT_AUTH_URL_UNPROTECTED`	Return the requested URL to the browser.
`CT_AUTH_BAD_PASSWORD` `CT_AUTH_BAD_USERNAME`	Return a `WWW-Authenticate` (HTTP 401) to the browser. For form-based authentication, the session is invalidated, and the user is redirected to a log on page.
`CT_SESSION_EXPIRED`	Force new user authentication.
`CT_AUTH_PASSWORD_EXPIRED_FORCED` `CT_AUTH_PASSWORD_EXPIRED_NEW_USER`	Force password change.

`CT_AUTH_EXPIRED_ACCOUNT`
`CT_AUTH_INACTIVE_ACCOUNT`
`CT_AUTH_PASSWORD_EXPIRED`
`CT_AUTH_URL_ACCESS_DENIED`
`CT_AUTH_USER_LOCKED_OUT`
	Return a `FORBIDDEN` (HTTP 403) to the browser.
	For form-based authentication, the user is redirected to the appropriate error page. The user is redirected to access denied page or the log on error page configured in *webagent.conf*.

`CT_AUTH_UNKNOWN_ERROR`
`CT_AUTH_DATABASE_ERROR`
`CT_COOKIE_ERROR`
`CT_NO_AUTH_SERVERS`
`CT_SERVER_TIMED_OUT`
`CT_UNHANDLED_REQUEST`
	Return an error message (HTTP/500) to the browser.
	For form-based authentication, the user is redirected to the appropriate error page. The user is redirected to access the denied page or the log on error page configured in *webagent.conf*.

Build a WAX Program

This section describes how to build a WAX program and integrate it into the Access Manager Agent processing loop.

High-Level Steps

A WAX program is built using the components of the WAX API. Assuming the Access Manager Agent is installed and integrated with the web server, adding a WAX program involves the following steps:

1. Use the `ct_wax_init()` methods to link each of the required WAX methods with one or more of the Access Manager Agent standard phase handlers.
2. Compile the WAX program. For more details, see "Compile and Link a WAX Program" on page 11.
3. Register the WAX program with the Access Manager Agent by adding its name to the `cleartrust.agent.wax` parameter in *webagent.conf*.

   If the WAX program provides user authentication, edit the `cleartrust.agent.auth_resource_list` in *webagent.conf* to specify an authentication type of `CUSTOM` for those resources that require authentication to be performed by the WAX methods.

WAX API Components

The WAX API is a C API that is installed as part of the Access Manager Agent installation. It consists of WAX API Headers and WAX API Libraries.

WAX API Headers

The WAX API header files are located in the `/include` directory of the Access Manager Agent installation. The WAX API consists of the following files:

- *ct_auth_result.h*
- *ct_external.h*
- *ct_function_table.h*
- *ct_memory.h*
- *ct_request_data.h*
- *ct_table.h*

WAX API Libraries

The WAX API works with the library file that is specific to the web server. The Access Manager Agent libraries are located at: `<AGENT_ROOT>\<web server>\lib\`

Compile and Link a WAX Program

These sections provide guidelines to compile and link the WAX API on the following platforms:

- Microsoft Internet Information Server
- Oracle Java System Web Server on UNIX
- Apache HTTP Server/ IBM HTTP Server on UNIX
- IBM Domino Web Server on Oracle Solaris.

Microsoft Internet Information Server

The following describes how to compile and link with the WAX API for Microsoft® Internet Information Server (IIS).

- For 32-bit:
  - Compiler: Microsoft Visual Studio 2005(VC8)
  - Compile option: `WIN32 _DEBUG, _WINDOWS MSIIS7,WINDOWS`
  - Additional Libraries: *ct_iis70_agent.lib*
- For 64-bit:
  - Compiler: Microsoft Platform SDK for AMD64
  - Compile option: `WP64, _DEBUG, _WINDOWS, MSIIS7, WINDOWS`
  - Additional Libraries: *ct_iis70_agent.lib*, *BufferoverflowU.lib*

Additional Include Directories: `<AGENT_ROOT>\include`
Additional Library Path: `<AGENT_ROOT>\lib`

Oracle Java System Web Server on UNIX

The following describes how to compile and link with the WAX API for Oracle® Java System Web Server on UNIX.

Compile options: `-DNETSCAPE -DFILE_UNIX -DXP_UNIX`

Additional include directories:

- `<AGENT_ROOT>/include`
- `<WEB_SERVER_DIR>/include`

Additional libraries: *libct_sjsws7_agent.so*
Additional library path: `<AGENT_ROOT>/lib`

Recommended link option: `-Bsymbolic`

The `-Bsymbolic` option forces the WAX to resolve the symbols against the API rather than the Access Manager Agent. If this is not used, name conflicts may be encountered with methods in the Access Manager Agent.

Where *gcc* is used to compile, the syntax is different and the recommended link options are: `Wl` and `Bsymbolic`.

Apache HTTP Server/ IBM HTTP Server on UNIX

The following describes how to compile and link with the WAX API for Apache™ or IBM® HTTP Server on UNIX.

Compile options: `-DAPACHE -DFILE_UNIX -DXP_UNIX`

Additional include directories:

- `<AGENT_ROOT>/include`
- `<HTTP_SERVER_DIR>/src/include`
- `<HTTP_SERVER_DIR>/src/os/unix`

Additional libraries: *libct_apache22_agent.so*
Additional library path: `<AGENT_ROOT>/lib`

Recommended link option: `-Bsymbolic`

The `-Bsymbolic` option forces the WAX to resolve the symbols against the API rather than the Access Manager Agent. If it is not used, name conflicts might be encountered with methods in the Access Manager Agent.

If using gcc, the recommended link options are: `Wl` and `Bsymbolic`.

IBM Domino Web Server on Oracle Solaris

The following describes how to compile and link with the WAX API for IBM Domino® Web Server on Oracle Solaris®.

Compile options: `-DFILE_UNIX -DXP_UNIX`

Additional include directories: `<AGENT_ROOT>/include`
Additional libraries: *libct_domino85_agent.so*
Additional library path: `<AGENT_ROOT>/lib`

Recommended link option: `-Bsymbolic`

The `-Bsymbolic` option forces the WAX to resolve the symbols against the API rather than the Access Manager Agent. If not used, name conflicts might be encountered with methods in the Access Manager Agent.

Write WAX Methods

Many required functionalities may be implemented inside the WAX methods. The WAX program can read the Access Manager Agent configuration parameters loaded from *webagent.conf*, which can include added custom parameters. For more details, see "Load Parameter Settings" on page 22.

- Write a WAX Method
- Register a WAX Method
- Invoke a WAX Method.

Write a WAX Method

A WAX method must return `TRUE` if it handled its phase, thereby omitting the default handler for that phase, or `FALSE` if it did not handle the phase. thereby allowing the default handler to run as soon as the WAX method returns. In summary:

- If the WAX method augments the handler with which it is associated, then it must return `FALSE`.
- If the WAX method replaces or overrides the handler with which it is associated, then it must return `TRUE`.

Register a WAX Method

The methods of the WAX program must be associated with one or more of the standard Access Manager phase handlers, so that the WAX methods are invoked when the standard phase handlers are called.

The phase handlers and the status handler are defined in a function table, which is a hash table consisting of handler keys and their associated function pointers. The keys define the various phase handlers that comprise the Access Manager Agent.

The list of handler keys and the function table structure are found in the *ct_function_table.h* header file, contained in the Access Manager Agent installation directory. For a description of each handler, see "Phase Handlers" on page 3.

To customize the action that results from a phase handler, or to alter the flow of URL request processing, use the `ct_table_put()` function call to register the WAX methods with phase handlers in the function table. Typically, registration of handlers is performed during initialization of the WAX program, inside the WAX `ct_wax_init()` method. The `ct_wax_init()` method must return 1 to indicate success.

In some cases, registration may be completed in a platform/Web server-specific method.

The following demonstrates the use of the `ct_wax_init()` method:

```
int ct_wax_init(ct_table_ptr ct_func_table, ct_table_ptr config)
{
ct_table_put(ct_func_table, CT_AUTHENTICATION_HANDLER,
my_custom_auth);
return 1;
}
```

When `ct_table_put()` is called, the arguments are, in order:

- `ct_func_table`. This is the function table passed in by the Access Manager Agent when it called `ct_wax_init()`.
- The handler key argument of the phase handler.
- The name of the WAX method to be associated with the phase handler.

The configuration can include multiple WAX methods. Multiple WAX methods may be associated with a single phase handler, and more than one phase handler may have WAX methods associated with it.

Important: If the `ct_print()` method is used in a WAX program, the statement must be formatted correctly, otherwise the web server could fail. In particular, do not pass in more format specifiers (for example, "%s" conversion flags) than there are parameters to fill them.

Register a WAX Authentication Method

A WAX authentication method **must** be registered with the `CT_AUTHENTICATION_HANDLER`. The custom authentication cannot be associated with any other handler.

To use the WAX authentication method, edit the *webagent.conf* file to specify a custom authentication type for all resources that require authentication by the WAX method.

- Add the names of these resources or directories of resources to the `cleartrust.agent.auth_resource_list` with their authentication type set to `CUSTOM`.

  Alternatively, set the `cleartrust.agent.default_auth_mode` to `CUSTOM` so that all resources use the custom WAX authentication, unless otherwise specified.

Invoke a WAX Method

WAX authentication methods will be called only when a user attempts to load a resource that is protected by the custom authentication type, `SC_AUTH_TYPE_CUSTOM`.

WAX non-authentication methods, such as a logging method, are called every time their associated phase handlers are called.

WAX API Reference

The WAX API reference section includes these topics:

- Initialization Functions
- Hash Table Functions
- Memory Management
- Print Status and Debugging Information
- Request Data
- Load Parameter Settings
- Load Parameter Settings
- Use WAX Programs with Virtual Host-Enabled Servers
- Backward Compatibility.

Initialization Functions

WAX programs loaded using the `cleartrust.agent.wax` directive must implement one or more of the WAX initialization functions. These functions are the WAX initialization entry points called by the Access Manager Agent on loading the WAX program.

There are several initialization functions, in order to properly handle multiprocess web servers such as Apache. On web servers that do not fork, the following functions are called in order. All of them share the same signature.

Function Signatures:

Code	Function
`int ct_wax_init(ct_table_ptr func_table, ct_table_ptr config)`	This is a required init function, called whenever the Agent is loaded by a web server process (both parent and child). Any initialization that is always performed must be done with this function. Configuration checking must be done with this function.
`int ct_wax_main_init(ct_table_ptr func_table, ct_table_ptr config)`	This is an optional function, called only in the main (parent) process before forking children to handle requests. Initialization that only needs to be performed by the parent, or time-consuming allocation of resources that can be shared with children, must be done using this function.
`int ct_wax_child_init(ct_table_ptr func_table, ct_table_ptr config)`	This is an optional function, called by each child process as it starts. Resources that cannot be shared between processes (such as sockets) must be allocated using this function.

Parameters:

- `func_table` is a pointer to the function table for registering handlers.
- `config` is a pointer to a table containing the Agent's configuration parameters, expressed as name/value pairs. These parameter settings are loaded from *webagent.conf*.

Return Value:

Returns a non-zero value on success. Otherwise it returns a 0 (zero).

Usage:

Define the `ct_wax_init()` method to call `ct_table_put` to add rows associating each WAX method with a phase handler. For an example, see "Register a WAX Method" on page 13.

Hash Table Functions

The Access Manager Agent uses hash tables for the function table (the list of phase handlers and, optionally, their WAX associations), for the configuration parameters loaded from *webagent.conf* and for the request data. The hash table functions are located in the *ct_table.h* header file.

The following table lists the hash table functions:

Code	Function
`void ct_table_put (ct_table_ptr table, const char* key, const void* value)`	Adds or replaces a value specified by the key. **Note:** The value is NOT copied. The pointer to the value is stored.
`void* ct_table_find(ct_table_ptr table, const char* key)`	Returns the value pointer for the key. If the key does not exist, a `NULL` is returned.
`void ct_table_remove(ct_table_ptr table, const char* key)`	Removes the value pointer for the key.
`void ct_table_replace (ct_table_ptr table, const char* key, const void* value);`	Replaces the current value in the table with the new value.

Memory Management

Use the Access Manager `ct_malloc()` and `ct_free()` methods to allocate or free memory on the Access Manager-protected web servers, rather than using the standard C methods. For complete information, consult the comments in *ct_function_table.h* and in c*t_memory.h*.

Code	Function
`void* ct_req_alloc (const ct_server_parms* server_parms, size_t size);`	Allocates memory which is freed automatically at the end of the request. Where possible, RSA recommends the use of `ct_req_alloc()` rather than `ct_malloc()`.
`char* ct_req_strdup (const ct_server_parms* server_parms,const char* str);`	Copies a string using memory which is freed automatically at the end of the request.
`void* ct_malloc(size_t size);`	Access Manager replacement for the standard `Cmalloc()` call. Where possible, RSA recommends the use of `ct_req_alloc()` rather than `ct_malloc()`. If using `ct_malloc()`, `ct_free()` must be called to free the memory before the WAX exits.
`void* ct_realloc(void* buf, size_text_size);`	Access Manager replacement for the standard C `realloc()` call.
`void ct_free(void* ptr);`	Access Manager replacement for the standard C `free()` call. Frees memory that was allocated using `ct_malloc()`. Not needed if using `ct_req_malloc()`.
`int ct_wax_cleanup(ct_table_ptr func_table, ct_table_ptr config);`	With dynamic restart functionality, the WAX must implement `ct_wax_cleanup(void)` to clean up memory. When the Access Manager Agent is restarted dynamically, it invokes this function and then reinitializes the WAX.
`char* ct_strdup(const char* str);`	Access Manager replacement for the standard C `strdup()` call.

Print Status and Debugging Information

Use the Access Manager `ct_print()` method to print status and debugging information. It is important to format the statement correctly, otherwise it could cause the web server to fail. In particular, do not pass in more format specifiers, for example `%s` conversion flags, than there are parameters to fill them.

Request Data

Data associated with a URL request is stored in a hash table called `ct_request_data`. This table is passed between phase handlers. The `ct_request_data` structure is located in the *ct_request_data.h* header file.

The `ct_request_data` hash table contains certain values that are constants and certain values that are dynamically allocated. If a WAX chooses to modify a dynamically allocated value, it must manage the memory associated with that value. When replacing a dynamic value, the old memory must be freed. When adding a dynamic value, it must be allocated using `ct_malloc` or `ct_strdup`.

The request data is passed to the phase handlers and is also directly available to functions other than phase handlers through the code:

```
ct_get_request_data_table (void* request)
```

The input parameter `request` is a pointer to the web server dependent structure, as in this request for IIS:

```
Filter Context structure, PHTTP_FILTER_CONTEXT
```

Note: For IIS, the `ct_request_data` is not available until the `SF_NOTIFY_URL_MAP` phase.

The following table indicates which values are dynamic. Request data is retrieved from the request data table through the hash lookup, using the keys described in the table.

Key	Type	Dynamic	Value
`CT_AGENT_USER`			
	char *	Yes	The user ID for the current session.
`CT_ALLOWABLE_AUTH_MODES`			
	char *	Yes	The allowable authentication types for the current request. The Authentication handler determines what types of authentication types are accepted for the current URL request: `BASIC`, `CERTIFICATE`, `CUSTOM`, `NT`, `SECURID`.

Key	Type	Dynamic	Value
CT_AUTHENTICATED			
	unsigned int	No	A bit mask that specifies the types of authentication the user is currently authenticated against: CT_AUTH_BASIC (0x00000001) CT_AUTH_CERTIFICATE (0x00000004) CT_AUTH_CUSTOM (0x00010000) CT_AUTH_NT (0x00000002) CT_AUTH_SECURID (0x00000008)
CT_AUTH_MODE			
	char *	No	• The Authorization mode. The Authorization handler determines what type of authorization to perform using this value. The values are: • UPW - Perform Authentication and Authorization. User ID and password are supplied. • UDN - Perform Authentication and Authorization. Locate user by Distinguished Name. • UID, UNT, CUSTOM, SECURID - Perform Authorization only. Locate user by user ID.
CT_DN			
	char *	Yes	The user Distinguished Name. When the CT_AUTH_MODE is set to UDN, this value is used to retrieve the Distinguished Name for authorization.
CT_ERR_MSG			
	char *	No	ct_err_msg stores the error message of the error that occurred during the processing of the URL. This message is returned to the client for error handling. The WAX developers may use this message to be displayed to the end user.
CT_FORM_AUTH_MODE			
	char *	Yes	The authentication mode of the form. This value specifies which authentication mechanism to use to authenticate the data in the form.
CT_IS_PATH_PROTECTED			
	char *	No	Specifies whether or not the URL is protected (Yes/No).
CT_RTAPI_POOL_INDEX			
	char *	Yes	Retrieves the Runtime API pool index. If the WAX needs to use the Runtime API, use this key to retrieve the pool index.

Key	Type	Dynamic	Value
CT_ORIG_URI			
	char *	No	Original URL requested before the user was redirected to a log on page.
CT_PASSWORD			
	char *	Yes	The user password. When the CT_AUTH_MODE is set to UPW, this value is used to retrieve the user's password for authentication.
CT_PLUGIN_USER			
	char *	Yes	The user ID. When the CT_AUTH_MODE is set to UPW, this value is used to retrieve the user ID for authentication. For both UPW and UID, this value is used for authorization. Deprecated since version 4.6. Instead use CT_AGENT_USER.
CT_POST_DATA			
	char *	No	Raw form data associated with a form-based log on request.
CT_PREV_USER			
	char *	Yes	The user ID or DN for the previous authentication.
CT_QUERY			
	char *	Yes	Query-string portion of the requested URL.
CT_RTAPI_POOL_INDEX			
	char *	Yes	Retrieves the Runtime API pool index. If the WAX needs to use the Runtime API, use this key to retrieve the pool index.
CT_STATUS			
	int	No	The status. This value is used by the status handler to determine the web server action and which handler to execute.
CT_URI			
	char *	Yes	The requested URL. If this value is overridden, the web server is instructed to provide the new URL.
CT_AGENT_USER			
	char *	Yes	The user ID for the current session.

Key	Type	Dynamic	Value
CT_USER_DATA			
	void *	No	A pointer to a buffer containing user-defined raw data included with the Access Manager cookie. The CT_USER_DATA must be a NULL terminated string.
CT_USER_DATA_LEN			
	unsigned short	No	The length of the CT_USER_DATA buffer, in bytes. The maximum length is 2048 bytes. Deprecated since version 4.6.

The web server can be configured to return a customized HTML page with the HTTP return codes.

Load Parameter Settings

When the Access Manager Agent calls `ct_wax_init()` to initialize the WAX program, it passes a pointer, the `config` parameter, to a table containing the Access Manager Agent configuration parameters, expressed as name/value pairs. These are the parameter settings loaded from the *webagent.conf* file.

The WAX program can use the `ct_table_find()` method to get the value of any parameter by name. Custom parameters can be added to and accessed from *webagent.conf*. For more details about `ct_table_find()`, see "Hash Table Functions" on page 17.

Each custom parameter added must begin with the prefix "`cleartrust`". RSA recommends naming WAX-related custom parameters with the prefix "`cleartrust.wax`" to distinguish them from standard Access Manager Agent parameters with the "`cleartrust.agent`" prefix.

Use WAX Programs with Virtual Host-Enabled Servers

To apply a WAX program to a single virtual host, declare the WAX in the `<VirtualHost ...>` block in *webagent.conf* for that virtual host. The same WAX program can be applied to many hosts by declaring it in the `<VirtualHost ...>` blocks for all hosts that use that WAX program.

A WAX declared in any `<VirtualHost ...>` block must not be declared in the `<Global>` block of *webagent.conf*.

When a WAX is running within the context of a virtual host, it has access to the *webagent.conf* parameters for that virtual host. These parameters include all the settings from the applicable `<VirtualHost ...>` block and, for each value not defined in that block, the setting from the `<Global>` block.

Backward Compatibility

This section describes any backward compatibility considerations.

Data Error Parameters Combined

Prior to RSA Cleartrust Agent 4.5, a group of duplicate error parameters for data errors existed in *ct_auth_result.h* and *ct_function_table.h*. These have been combined into a single set.

Duplicate Error Parameters Prior to RSA Cleartrust Agent 4.5

```
CT_LDAP_AUTH_ERROR = 50,
CT_UNKNOWN_ERROR = 100,
CT_DATABASE_ERROR = 101,
#define CT_LDAP_AUTH_UNKNOWN_ERROR 50
#define CT_AUTH_UNKNOWN_ERROR 100
#define CT_AUTH_DATABASE_ERROR 101
```

Duplicate Error Parameters Combined in RSA Cleartrust Agent 4.5

```
CT_LDAP_AUTH_ERROR = 50,
CT_AUTH_UNKNOWN_ERROR = 100,
CT_AUTH_DATABASE_ERROR = 101,
```

Modify existing extensions to account for these changes to *ct_auth_result.h* and *ct_function_table.h.*

WAX API Examples

The following examples show how to write and register a WAX program. Example programs are included in the product file. The WAX API example code can be retrieved and installed from the `zip` file.

- Cookie Data Example
- Custom Authentication Example
- URL Redirect Example.

Cookie Data Example

The following steps describe how to add data to the Access Manager cookie.

1. At startup, the Access Manager Agent checks the `cleartrust.agent.wax` parameter in its *webagent.conf* file and loads all the WAX programs listed there. n the example provided, the parameter could be set similar to the following:

   ```
   cleartrust.agent.wax=D:\\wax_programs\\lib\\wax.so
   ```

2. The Access Manager Agent calls the `ct_wax_init()` method in the WAX program and associates the WAX methods with the Access Manager Agents phase handlers in the function table, `ct_func_table` . When those phase handlers run, they automatically call the associated methods in the WAX.

   In the example provided, there is just one WAX method, `my_cookie_phase_handler()`, and it is associated with the authentication handler, `CT_COOKIE_HANDLER`.

3. When a WAX method runs, it sets the status and returns `TRUE` if it handled this phase, thereby omitting the default handler. It returns `FALSE` if it did not handle the phase. For more details, see "Write WAX Methods" on page 13.

wax.c is a WAX example provided to show how to add data to the Access Manager cookie. In this example, the WAX merely augments the `CT_COOKIE_HANDLER`, so it returns `FALSE`, indicating that the cookie handler must still run.

The following is the *wax.c* example:

```
/*
* wax.c
*
* This example, wax.c, is a Web Agent Extension (WAX) that
* shows how to insert data into the RSA ClearTrust SSO token.
*/
// The standard includes
#include <stdio.h>
#include <string.h>
// Only include windows.h on Windows
#ifdef _WINDOWS
#include <windows.h>
#endif
// ClearTrust includes
#include "ct_function_table.h"
#include "ct_request_data.h"
#include "ct_external.h"
```

```
// Internal macros
#define SUCCESS 1
#define FAILURE 0
#define EMAIL "foo@bar.com"
#define EMAIL_LENGTH strlen(EMAIL)
// Prototypes declaration
/**
* This function is the initial interface between ClearTrust Agent
* and the Web Agent eXtension (WAX).
*/
CT_EXTERNAL int ct_wax_init (ct_table_ptr, ct_table_ptr);
/**
* The "Cookie phase handler". This function is invoked by ClearTrust
* Agent before the ClearTrust Cookie Phase is handled.
*/
CT_EXTERNAL int my_cookie_phase_handler (const ct_server_parms *,
ct_table_ptr);

/**
* This function is the initial interface between ClearTrust Agent
* and the Web Agent eXtension (WAX). The function registers the custom
* phase handlers. In this case, we are registering a custom cookie
* phase handler.
*/
CT_EXTERNAL int ct_wax_init (ct_table_ptr ct_func_table,
ct_table_ptr configuration)
{
ct_print ("ct_wax_init is invoked ...\n");
ct_table_put (ct_func_table,
CT_COOKIE_HANDLER,
my_cookie_phase_handler);
return SUCCESS;
} // End of ct_wax_init
/**
* The "Cookie phase handler". This function is invoked by ClearTrust
* Agent before the ClearTrust Cookie Phase is handled.
*/
CT_EXTERNAL int my_cookie_phase_handler (const ct_server_parms *
server_parms,
ct_table_ptr ct_request_table)
{
ct_print ("my_cookie_phase_handler is invoked ...\n");
ct_table_put (ct_request_table, CT_USER_DATA, (void *) EMAIL);
ct_table_put (ct_request_table, CT_USER_DATA_LEN, (void *)
EMAIL_LENGTH);
return FAILURE;
} // End of my_cookie_phase_handler
```

Custom Authentication Example

The following steps describe how a WAX registers and uses its authentication handler:

1. At startup, the Access Manager Agent checks the `cleartrust.agent.wax` parameter in its *webagent.conf* file and loads all the WAX programs listed there. In the following example, the parameter is set as follows:

   ```
   cleartrust.agent.wax=D:\\wax_programs\\lib\\nt_auth.so
   ```

2. The Access Manager Agent calls the `ct_wax_init()` method in the WAX program and associates the WAX methods with the authentication handlers in the function table, `ct_func_table`.

3. When the authentication handler runs, it calls the associated WAX method only if the requested resource (URL) is protected by the custom authentication type. For details, see "Invoke a WAX Method" on page 14.

   In the example provided, there is just one WAX method, `nt_authenticate`, and it is associated with the authentication handler `CT_AUTHENTICATION_HANDLER`.

4. When a WAX method runs, it sets the status and returns `TRUE` if it handled this phase, thereby omitting the default handler, or it returns `FALSE` if it did not handle the phase. For more details, see "Write a WAX Method" on page 13.

   In the following example, the WAX handles the authentication phase if a user name and password is provided. In order for this example to work, the user names of the registered users in Access Manager must be identical to their Windows NT user names.

nt_auth.c is a WAX example provided to show how to add a custom authentication routine. This example replaces the Access Manager authentication with native Windows NT authentication.

Note: This example uses NT authentication for purposes of demonstration only. Such a WAX program would not actually be built, because NT authentication is a standard feature of Access Manager.

During web server initialization, the *nt_auth.c* WAX registers its authentication handler in the function table. When the authentication handler is invoked, the *nt_auth.c* WAX checks whether or not a User ID and password have been set. If they have been set, the WAX performs NT native authentication, sets the status, and returns `TRUE`, which indicates that it has succeeded in handling the authentication phase.

In this example, there is just one WAX method, `nt_authenticate()`, and it is associated with the authentication handler `CT_AUTHENTICATION_HANDLER`.

The following is the *nt_auth.c* example:

```
/*
* nt_auth.c
*
* This example, nt_auth.c, is a Web Agent Extension (WAX) that
* shows how to add a custom authentication routine. This
* example replaces the Access Manager authentication with
* native NT authentication. This example uses NT authentication
* for purposes of demonstration. Such an extension would not
* actually be built, since NT authentication is a standard
* feature of the RSA ClearTrust system.
*/
#include <stdio.h>
// Windows header files
#include <windows.h>
#include <winnt.h>
// ClearTrust header files
#include "ct_auth_result.h"
#include "ct_function_table.h"
#include "ct_request_data.h"
#include "ct_external.h"
// Prototype for the NT authentication
CT_EXTERNAL int nt_authenticate(const ct_server_parms *server_parms,
ct_table_ptr ct_req_table);
/**
* Initialization method for Web Agent Extension.
*/
CT_EXTERNAL int ct_wax_init(ct_table_ptr ct_func_table,
ct_table_ptr conf)
{
ct_table_put(ct_func_table,
CT_AUTHENTICATION_HANDLER,
nt_authenticate);
return 1;
}
/*
* Routine to perform NT authentication. It first calls the
* LogonUser API to perform NT authentication, then sets the
* appropriate status. This routine returns TRUE only if the user
* and password is set. If the user and password isn't set,
* it lets the default authentication execute which prompts the
* user for the user and password.
* When the user and password is set, the NT authentication WAX
* makes the NT API call authenticating the user and then
* returns a TRUE preventing default authentication.
*/
CT_EXTERNAL int nt_authenticate(const ct_server_parms *server_parms,
ct_table_ptr ct_req_table)
{
BOOL bHandled = FALSE; // If no user or pw, then don't handle
BOOL bIsAuthenticated = FALSE;
HANDLE hToken = 0;
LPTSTR lpszUser = ct_table_find(ct_req_table, CT_PLUGIN_USER);
LPTSTR lpszPassword = ct_table_find(ct_req_table,
CT_PASSWORD);
// If the username and password are supplied, we'll handle
// authentication.
if (lpszUser!= NULL && lpszPassword!= NULL)
{
// Perform NT authentication. We specify a NULL domain name
// so the User will be searched throughout all the PDCs.
// Also, we call the NT method LogonUser() with
// LOGON32_LOGON_NETWORK logon type because we are authenticating
// the user, not creating a process under the User's account.
bIsAuthenticated = LogonUser(lpszUser,
NULL,
lpszPassword,
LOGON32_LOGON_NETWORK,
LOGON32_PROVIDER_DEFAULT,
```

```
&hToken);
// If isAuthenicated isn't 0, then the user is authenticated.
if (bIsAuthenticated)
{
ct_table_put(ct_req_table,
CT_AUTHENTICATED,
(void *) CT_AUTH_CUSTOM);
// Force access checking
SET_STATUS(ct_req_table, CT_CHECK_ACCESS_REQUIRED);
}
else
{
DWORD dwError = GetLastError();
// Set the appropriate ClearTrust Error code
switch(dwError)
{
case ERROR_LOGON_FAILURE:
SET_STATUS(ct_req_table, CT_AUTH_BAD_USERNAME);
break;
default:
SET_STATUS(ct_req_table, CT_AUTH_UNKNOWN_ERROR);
break;
}
}
bHandled = TRUE; // This indicates that we have handled the
// authentication stage and the Agent can proceed directly
// to the status handler.
}
return bHandled;
}
```

URL Redirect Example

The WAX API can be used to redirect a user from the requested URL to a new URL. This example demonstrates how to return a custom page when a user is denied access to an Access Manager-protected resource. The *redirect.c* WAX program shows how to replace the requested URL with a new URL to display an error code that Access Manager returns.

During web server initialization, the WAX programs registers its status handler in the function table. When the status handler is invoked, it checks the current status. If the status is an error, it replaces the requested URL with a corresponding custom error page and sets the status to `CT_AUTH_URL_ACCESS_ALLOWED`, that is, access is allowed for the error page only. This forces the Access Manager Agent to provide the new URL.

The following is the *redirect.c* example:

```
/*
* redirect.c
*
* Use the WAX API to return a custom page when a User is
* denied access to an Access Manager-protected resource.
* This WAX example shows how to replace the requested URI with
* a new URI to display to the user an Access Manager error code.
*
*/
#include <stdio.h>
// Windows header files
#include <windows.h>
#include <winnt.h>
// ClearTrust header files
#include "ct_auth_result.h"
#include "ct_function_table.h"
#include "ct_request_data.h"
#include "ct_external.h"
// Prototype for status handler
CT_EXTERNAL int handle_status(const ct_server_parms *server_parms,
ct_table_ptr
ct_req_table);
// Module Definitions for the customer error pages
#define BAD_USER_PAGE "/cleartrust/bad_user.html"
#define INVALID_ACCOUNT_PAGE "/cleartrust/invalid_account.html"
#define USER_FORBIDDEN_PAGE "/cleartrust/forbidden_user.html"
/**
* Initialization method for Web Agent Extension.
*/
CT_EXTERNAL int ct_wax_init(ct_table_ptr ct_func_table, ct_table_ptr
conf)
{
ct_table_put(ct_func_table, CT_STATUS_HANDLER, handle_status);
return 1;
}
/**
* Status Handler. Checks for the error codes which we want to return a
* custom error page. If we are returning a custom error page, then
* set the return code to TRUE indicating that we handled the status.
* Also, set Status to CT_AUTH_URL_ACCESS_ALLOWED forcing the serving
* of the custom error page.
*/
CT_EXTERNAL int handle_status(const ct_server_parms *server_parms,
ct_table_ptr ct_req_table)
{
BOOL bHandled = FALSE;
```

```
// Switch off the current status
switch(GET_STATUS(ct_req_table))
{
// If we have a bad user name, then we have an un-registered
// user. Serve up registration page.
case CT_AUTH_BAD_USERNAME:
// Since BAD_USERNAME is returned if no User Name has been
// supplied, we only want to redirect if one is supplied
if (ct_table_find(ct_req_table, CT_PLUGIN_USER)!= NULL)
{
ct_table_put(ct_req_table, CT_URI, BAD_USER_PAGE);
SET_STATUS(ct_req_table, CT_AUTH_URL_ACCESS_ALLOWED);
bHandled = TRUE;
}
break;
// If we have an expired or inactive account, then we need
// to re-register the user
case CT_AUTH_EXPIRED_ACCOUNT:
case CT_AUTH_INACTIVE_ACCOUNT:
ct_table_put(ct_req_table, CT_URI, INVALID_ACCOUNT_PAGE);
SET_STATUS(ct_req_table, CT_AUTH_URL_ACCESS_ALLOWED);
bHandled = TRUE;
break;
// If the user is denied, serve up a custom error page.
case CT_AUTH_URL_ACCESS_DENIED:
ct_table_put(ct_req_table, CT_URI, USER_FORBIDDEN_PAGE);
SET_STATUS(ct_req_table, CT_AUTH_URL_ACCESS_ALLOWED);
bHandled = TRUE;
break;
default:
break;
}
return bHandled;
}
```

RSA Support and Service

Read this section if you wish to contact RSA or request technical support or services.

RSA Customer Support

Support Contacts

Access these locations for help with your RSA product.

- RSA SecurCare Online

  RSA SecurCare Online offers a knowledge base that contains answers to common questions and solutions to known problems. It also offers information on new releases, important technical news, and software downloads.

- Customer Support Information

  The RSA Customer Support Information site contains information on RSA support programs plus an extensive Content Library of product-related documents such as datasheets, guides and whitepapers.

- RSA Secured Partner Solutions Directory

  The RSA Secured Partner Solutions Directory provides information about third-party hardware and software products that have been certified to work with RSA products. The directory includes Implementation Guides with step-by-step instructions and other information about interoperation of RSA products with these third-party products.

Before You Call Customer Support

Make sure you have direct access to the computer running your RSA product software. Please have the following information available when you call:

- ❑ Your RSA Customer/License ID.
  This is a paper license. You can find this number only on the license distribution medium. If you do not have access to the paper-based RSA Customer/License ID, contact RSA Customer Support.
- ❑ The software version number of your RSA product.
- ❑ The make and model of the machine on which the problem occurs.
- ❑ The name and version of the operating system under which the problem occurs.

Contact RSA

RSA is the premier provider of security solutions for business acceleration. The RSA technology, business and industry solutions — coupled with professional services and third-party strategic partnerships — help customers put critical information into the hands of the people who need it, while protecting that information against unauthorized access.